Escaping Groundhog Day

Embedding Business Objects in an Evolving Landscape of Cross-Cutting Concern Utilities

Herwig Mannaert

October, 2020

Universiteit Antwerpen
Embedding Business Objects in an Evolvable Landscape of Cross-Cutting Concern Utilities

- Groundhog Day
- Foundations of Evolvable Software
- Toward Scalable Metaprogramming
- A Glimpse Beyond Software
- Conclusion
Embedding Business Objects in an Evolvable Landscape of Cross-Cutting Concern Utilities

Overview

• Groundhog Day
  • Distributed Business Services

• Foundations of Evolvable Software
• Toward Scalable Metaprogramming
• A Glimpse Beyond Software
• Conclusion
Groundhog Day

• Popular American tradition
  • Mainly in Pennsylvania (*Punxsutawney*)
  • Based on groundhog emerging from its burrow
• Fantasy comedy from 1993
  • Man becomes trapped in a time loop
  • Forced to relive February 2 over and over again
• Became part of the English lexicon
  • Monotonous, unpleasant, and repetitive situation
  • *Series of unwelcome or tedious events appear to be recurring in exactly the same way*
The Quest for Distributed Plug & Play

• Monolithic applications dominated 1960’s and 1970’s
• Distributed architectures & standards emerged from 1980’s:
  • DCE/RPC in 1980’s – 1990’s
  • CORBA in 1990’s – 2000’s
  • XML/RPC – Web Services in 2000’s – 2010’s
  • JSON/RPC – REST Services in 2010’s – 2020’s
  • Service Mesh – Sidecar Proxy in 2020’s ...
• Business objects and capabilities
  • are (re-)implemented in a repetitive way
  • in an often tedious and recurring way
XML/RPC – Web Services

• Participants
  • Service Provider
  • Service Requester
  • Service Broker

• Protocols
  • SOAP: HTTP – XML
  • WSDL
  • UDDI
JSON/RPC – REST Services

• Participants
  • Service Provider
  • Service Requester
  • Service Broker

• Protocols
  • HTTP – JSON
  • Swagger
  • URI
Common Object Request Broker Architecture

- **Participants**
  - Client Requester
  - Server Provider
  - Naming Service

- **Protocols**
  - IIOP
  - IDL
  - COS
Java Remote Method Invocation

• Participants
  • Client Requester
  • Server Provider
  • Naming Service

• Protocols
  • RMI
  • Java IDL
  • JNDI
DCE / Remote Procedure Call

- **Participants**
  - Server
  - Client
  - Directory Service

- **Protocols**
  - DCE
  - IDL
  - UUID
Software as a Service

• New service infrastructures emerge, based on:
  • Cloud Computing
  • Containerization
  • Serverless
  • Datastores
  • Service Mesh
  • Side-car Proxy
  • …

• But basically …
SaaS Platforms – Some Recent Cases

Local government

HR Company
About the Tedious and Recurring Nature

- The business logic is not cleanly separated from the middleware
- Communication middleware protocol is not the only bus dimension:
  - Authorization
  - Access control
  - Authentication
  - Load balancing
  - Logging, archiving
  - ...
- All these concerns are intertwined with business logic, causing:
  - Duplication of cross-cutting concerns
  - Duplication of business logic
i.e., the Enterprise Service Bus Fallacy ...

Peer to Peer | Vs. | Integration Bus
---|---|---
\( \frac{N(N-1)}{2} \) | # links total | N
\( N \) | # links / node | 1
2 | # concerns / link | 1

We need multiple ESB’s, i.e., for every concern

The encapsulations do not realize the decoupling
Embedding Business Objects in an Evolvable Landscape of Cross-Cutting Concern Utilities

- Groundhog Day
- Foundations of Evolvable Software
  - Stability
  - Regeneration
  - Meta-Circularity
- Toward Scalable Metaprogramming
- A Glimpse Beyond Software
- Conclusion
Design Theorems for Stable Software

• In order to avoid dynamic instabilities in the software design cycle, the *rippling of changes needs to be depleted or damped: $a = 0$*

• As these ripples create *combinations of multiple changes* for every functional change, we call these instabilities *combinatorial effects*.

• Demanding systems theoretic stability for the software transformation, leads to the derivation of **principles** in line with existing heuristics.

• Adhering to these principles avoids dynamic instabilities, meaning that these **principles are necessary, not sufficient for systems stability**.
Change Ripples: A Basic Transformation

$D_m$
Invoice
- Number
- Order
- ...

$S_m = I(D_m)$
Invoice
- Number
- Order
- ...

$P_n$
CreateInvoice
ProcessOrder
SendInvoice

CreateInvoice
Add Attribute

ProcessOrder

SendInvoice

IMPACT

F_n = I(P_n)
CreateInvoice
ProcessOrder
SendInvoice

IMPACT

IMPACT

IMPACT
Encapsulating Basic Primitives

Invoice
  -Number
  -Order
  ...

Order
  -Ref
  -Product
  ...

CreateInvoice

ProcessOrder

SendInvoice
Separating Cross-Cutting Concerns
The Emergence of Elements
An Advanced Transformation

\[ I(D_m, T_\alpha) = \{S_{m,k}\}_{k=1,\ldots,K} \cup \{F_{m,l}\}_{l=1,\ldots,L} \]

- Invoice Nr
- Invoice Date

- Remote Access
- Persistency
- Access Control
- REST Connector
- SOAP Connector
- SOAP Connector
An Advanced Transformation
Element structures are needed to interconnect with CCC solutions.

NS defines 5 types of elements, aligned with basic software concepts:

- **Data elements**, to represent data variables and structures
- **Task elements**, to represent instructions and/or functions
- **Flow elements**, to handle control flow and orchestrations
- **Connector elements**, to allow for input/output commands
- **Trigger elements**, to offer periodic clock-like control

It seems obvious to use code generation techniques to create instances of these recurrent element structures.

Due to its simple and deterministic nature, we refer to this process as expansion, and to the generators as expanders.
Expansion of Elements

Invoice
-Number
-Order
...

Order
-Ref
-Product
...

CreateInvoice
SendInvoice
ProcessOrder

Conn. Element
Data Element
Task Element
Flow Element
Trigger Element

Order-Ref-Product...
Invoice-Number-Order...

ProcessOrder

Order - Ref - Product...
Invoice - Number - Order...

CreateInvoice
SendInvoice
Embedding Business Objects in an Evolvable Landscape of Cross-Cutting Concern Utilities

- Groundhog Day
- Foundations of Evolvable Software
  - Stability
  - Regeneration
  - Meta-Circularity
- Toward Scalable Metaprogramming
- A Glimpse Beyond Software
- Conclusion
On Updating Recurring Structure

• Structure should be recurring, as variations:
  • increase complexity of codebase
  • decrease consistency in behaviour

• Recurring structure may need to vary over time:
  • new insights
  • discovery of flaws
  • changes in technologies

Structural changes may need to be applied with retroactive effect, but the efforts increase with the frequency of change.

\[
N \text{ instances, update every } K \implies \# \text{updates} = \frac{N(N+K)}{2K}
\]
Recurrent stable structures are required to limit complexity and to guarantee consistency.

Recurrent stable structures need to be able to adapt over time, to overcome flaws and technology changes.

Additional custom code is inevitable and needs to be maintained across updated stable structures.

An automated mechanism is required, providing both code generation or expansion, and regeneration with harvesting and injection.

Catch 22: The Only Way Out
Variability Dimensions and Expansion

• We identify four dimensions of variability:
  • Models or *mirrors*, new data attributes/relations, new elements
  • Expanders or *skeletons*, new or improved implementations of concerns
  • Infrastructure or *utilities*, new frameworks to implement various concerns
  • Custom code or *craftings*, new or improved implementations of tasks, screens

• *If separated and well encapsulated*
  • Number of versions to maintain is *additive*: \#V = \#M + \#E + \#I + \#C
  • Number of versions available is *multiplicative*: \#V = \#M \times \#E \times \#I \times \#C
  • Where the same holds within any individual dimensions,
    e.g., infrastructure dimension: \#I = \#G \times \#P \times \#B \times \#T
Integrating the Dimensions of Variability
Change Dimension 1: The Mirrors

Mirrors

Order
-Ref
-Product

ProcessOrder

Invoice
-Number
-Order

CreateInvoice

SendInvoice

Skeletos

Utilities

Access Control

Persistency

Transaction

Craftings

Codebase
Change Dimension 2: The Utilities

- Mirrors
- Skeletons
- Utilities
  - Access Control
  - Persistency
- Transaction

Craftings

Codebase
Change Dimension 3: The Skeletons

MIRRORS
- Order
  - Ref
  - Product
- ProcessOrder
- CreateInvoice
- SendInvoice

SKELETONS

CRAFTINGS

CODEBASE

UTILITIES
- Access Control
- Persistency

TRANSACTION

The Skeletons

Codebase

Utilities
Change Dimension 4: The Craftings

Mirrors

Order
  - Ref
  - Product
  ...

ProcessOrder

CreateInvoice

SendInvoice

Skeletons

Invoice
  - Number
  - Order
  ...

Craftings

Codebase

Utilities

Access Control

Transaction

Persistency
Change Dimension 4: The Craftings

- Mirrors
  - Order
    - Ref
    - Product
  - ProcessOrder
  - CreateInvoice
  - SendInvoice

- Skeletons
  - Invoice
    - Number
    - Order

- Craftings
  - Feature 1
  - Feature 2
  - Feature N

- Codebase

- Utilities
  - Access Control
  - Persistency

- Transaction
Sustaining an Evolving Utility Landscape

**Ex Ante 8** A technology implementation of a specific concern for one element, or a listed set of elements, can be changed in a stable way.

- Remarks:
  - Part of $S_{\text{marg}}$:
    - $\forall$ artifact | technology = utility
    - One for every listed element
  - Craftings: no direct utility calls

\[
S_{\text{marg}} \subset \{S_{m,k}\}_{k=1,...,K} \cup \{F_{m,l}\}_{l=1,...,L}
\]
Ex Ante 9 – Expansion  An additional technology implementation for a specific concern of a type of element, can be made available for all information systems in a stable way.

• Remarks:
  • Part of $S_{marg}$:
    • $\forall$ template $|$ technology = utility
  • Configuration:
    • Define setting or option
  • Craftings: no direct utility calls
Sustaining an Evolving Utility Landscape

**Ex Ante 11 – Expansion** An additional concern for an of element can be made available for all information systems in a stable way.

- Remarks:
  - Part of $S_{marg}$:
    - $\{ templates \mid concern = concern \}$
  - Configuration:
    - Define setting or option
Embedding Business Objects in an Evolvable Landscape of Cross-Cutting Concern Utilities

Overview

• Groundhog Day
• Foundations of Evolvable Software
  • Stability
  • Regeneration
  • Meta-Circularity
• Toward Scalable Metaprogramming
• A Glimpse Beyond Software
• Conclusion
Meta-Circularity in Software Engineering

• Associated with terms like
  • *Homoiconicity*, coined in 1965, by Mooers & Deutsch (TRAC), uses concepts like “code as data” and “program structure similar to its syntax
  • *Meta-Circular Evaluator*, coined by John Reynolds in 1972, for an interpreter defining each feature of the defined language by using the corresponding feature of the defining language.

• Believed to *increase the abstraction level and therefore the productivity*

• Notion seems quite fundamental:
  • A transistor is switched by a transistor
  • A cell is produced by a cell
On Meta-Circularity in Metaprogramming

• You also have to maintain the meta-code
  • Consists of several modules
  • Is in general not trivial to write

• Will face growing number of implementations:
  • Different versions
  • Multiple variants
  • Various technology stacks

• Will have to adapt itself to:
  • Evolutions of its underlying technology
    • Which even may become obsolete

• **Meta-Circularity: meta-code that (re)generates itself**
Establishing the Meta-Circle: Phase 1

- Reader classes
- Model classes
- Control classes
- Expander classes
- Code Templates

Diagram:

- Model
- Read / Write
- Logic
- Control
- View

Expand
Application
Establishing the Meta-Circle: Phase 2

Prime Radiant

Expand

Expand

Application

Read / Write
Model
Logic
Control
View

nsx-prime

Reader classes
Model classes
Control classes
Expander classes

Code Templates

Meta Model
Model

Prime Radiant

Expand

Expand

Application

Read / Write
Model
Logic
Control
View
Establishing the MetaCircle: Phase 3

Prime Radiant → nsx-prime

Meta Model → Model → Reader classes → Control classes → Expander classes → Code Templates

Model classes

Logic Control View

Read / Write Model Logic Control View

Application Expand Expand
Establishing the MetaCircle: Phase 3

Expander Model

Meta Model

Model

Read / Write

Model

Logic

Control

View

Expander Model

MetaCircle

Code Templates

Prime Radiant

Expand

Expand

Application

Read / Write

Model

Logic

Control

View
Embedding Business Objects in an Evolvable Landscape of Cross-Cutting Concern Utilities

Overview

- Groundhog Day
- Foundations of Evolvable Software
- Toward Scalable Metaprogramming
  - Horizontal Integration
  - Realizing a Meta-ESB
- A Glimpse Beyond Software
- Conclusion
On Meta-Circularity in Meta-Programming

• You also have to maintain the meta-code
  • Consists of several modules
  • Is in general not trivial to write
• Will face growing number of implementations:
  • Different versions
  • Multiple variants
  • Various technology stacks
• Will have to adapt itself to:
  • Evolutions of its underlying technology
    • Which even may become obsolete
• **Meta-Circularity:** meta-code that (re)generates itself
Vertical Integration or Metaprogramming Silos

MP\textsubscript{1}

Model

Reader classes

Model classes

Control classes

Generator classes

Code Templates

Source Code

MP\textsubscript{2}

Model

Reader classes

Model classes

Control classes

Generator classes

Code Templates

Source Code

\ldots

MP\textsubscript{n}

Model

Reader classes

Model classes

Control classes

Generator classes

Code Templates

Source Code
Vertical Integration or Metaprogramming Silos

MP_1
  Model
  Reader classes
  Model classes
  Control classes
  Generator classes
  Code Templates
  Source Code

MP_2
  Model
  Reader classes
  Model classes
  Control classes
  Generator classes
  Code Templates
  Source Code

... MP_n
  Model
  Reader classes
  Model classes
  Control classes
  Generator classes
  Code Templates
  Source Code
To Horizontal Integration in Metaprogramming
On Horizontal Integration in Metaprogramming

Models

$\text{Model}_1$ $\text{Model}_3$ $\ldots$ $\text{Model}_N$

$\text{Model}_2$ $\text{Model}_4$ $\ldots$ $\text{Model}_N$

$\times$

Templates

$\text{Code Templates}_1$ $\text{Code Templates}_2$ $\text{Code Templates}_3$ $\ldots$ $\text{Code Templates}_M$

Source

Source Code $\times$ Source Code $\times$ $\ldots$ $\times$ Source Code

$N \times M$
Embedding Business Objects in an Evolvable Landscape of Cross-Cutting Concern Utilities

Overview

• Groundhog Day
• Foundations of Evolvable Software
• Toward Scalable Metaprogramming
  • Horizontal Integration
  • Realizing a Meta-ESB
• A Glimpse Beyond Software
• Conclusion
Metaprogramming Normalized Systems – Architecture

- Define meta-model
- Extend meta-model

- Meta-model
- Model

- Read / Write
- Model
- Logic
- Control
- View

- Expand meta-model
- Provide templates

- Prime Radiant
- Expand

- Expander Model
- MetaCircle

- Define expanders
- Code Templates

- Application
Integrating and Activating new Meta-Models

• Creation of a metaprogramming bus:
  • based on an horizontal integration architecture
  • using XML to exchange between models and templates
• Normalized Systems environment allows to:
  • define any Entity Relationship Diagram (ERD), including entities and relationships, e.g., Service, Cloud
  • generate the meta-circular stack for these entities, including:
    • XML readers and writers, e.g., ServiceXmlReader, ServiceXmlWriter
    • classes representing model instances, e.g., ServiceDetails, ServiceComposite
    • view and control classes for create and manipulate models in a user interface
  • make the models available to the templates through Object-Graph Navigation Language (OGNL) expressions
    • e.g., service.name, service.cloud.name, service.cloud.provider
Artifact = Expansion(Template, Model)
Embedding Business Objects in an Evolvable Landscape of Cross-Cutting Concern Utilities

Overview

• Groundhog Day
• Foundations of Evolvable Software
• Toward Scalable Metaprogramming
• A Glimpse Beyond Software
  • Embedding Evolvable Utilities
• Conclusion
Guidelines on Cross-Cutting Concern

- Encapsulation
- Interconnection
- Downpropagation
A Basic Example: Heating

- **Encapsulation**
  - Fireplace $\rightarrow$ electric heater

- **Interconnection**
  - Fireplaces, electric heaters $\rightarrow$ central heating system
  - Central heating system $\rightarrow$ district and city heating

- **Downpropagation**
  - Central heating system $\rightarrow$ radiators in rooms
  - District and city heating $\rightarrow$ individual houses $\rightarrow$ rooms
  - *Can we propagate down to the individual “business objects”?*
Some Construction Concept Elements
Embedding Business Objects in an Evolvable Landscape of Cross-Cutting Concern Utilities

Overview

• Groundhog Day
• Foundations of Evolvable Software
• Toward Scalable Metaprogramming
• A Glimpse Beyond Software
• Conclusion
Conclusion

• We have indicated that the development of distributed landscapes of business functions may exhibit some tedious recurrent behavior

• Based on a thorough analysis, we have argued that evolvable software requires structured and preferably meta-circular metaprogramming

• We have explained how this meta-circular environment enables
  • the creation of dedicated meta-models to define parameters
  • to generate the infrastructure code to embed business objects in services
  • and to regenerate them in evolving cross-cutting concern landscapes

• While the metaprogramming environment has been successfully applied for years to (re)generate web-based information systems, the use of custom meta-models to generate service encapsulations is just starting
Some References

- Mannaert Herwig, McGroarty Chris, Gallant Scott, De Cock Koen, **Integrating Two Metaprogramming Environments : An Explorative Case Study** : ICSEA 2020 - ISSN 2308-4235 - IARIA, 2020, p. 166-172
- Mannaert Herwig, Verelst Jan, Ven Kris, **The transformation of requirements into software primitives : studying evolvability based on systems theoretic stability**, Science of computer programming - ISSN 0167-6423 - 76:12(2011), p. 1210-1222
- **Normalized Systems Foundation Lectures** : https://www.youtube.com/channel/UCc8P1LREJogSlhwmAvdkq2A
QUESTIONS ?
herwig.mannaert@uantwerp.be